用单子描述计算

1 单子的定义

每一个不同的单子都用于描述一类计算：

- 假设 M 是一个单子，
- $f \in M(A)$ 表示 f 是一个计算结果（或返回值）的类型为 A 的计算，
- M 单子应配有 bind 算子表示计算的复合，即 bind: $\forall AB.\ M(A) \to (A \to M(B)) \to M(B)$；
- M 单子应配有 return 算子表示直接返回一个特定值，即 return: $\forall A.\ A \to M(A)$。

2 在 Coq 中定义单子结构

单子可以这样在 Coq 中定义为一类代数结构。

```
Class Monad (M: Type -> Type): Type := {
  bind: forall {A B: Type}, M A -> (A -> M B) -> M B;
  ret: forall {A: Type}, A -> M A;
}.
```

我们之后最常常用到的将是集合单子（set monad）如下定义。

```
Module SetMonad.

Definition M (A: Type): Type := A -> Prop.

Definition bind: forall (A B: Type) (f: M A) (g: A -> M B), M B :=
  fun (A B: Type) (f: M A) (g: A -> M B) =>
    fun b: B => exists a: A, a ∈ f /\ b ∈ g a.

Definition ret: forall (A: Type) (a: A), M A :=
  fun (A: Type) (a: A) => Sets.singleton a.

End SetMonad.
```

```
#[export] Instance set_monad: Monad SetMonad.M := {|
  bind := SetMonad.bind;
  ret := SetMonad.ret;
|}.
```

下面是另一个例子状态单子的定义：

```
Module StateMonad.

Definition M (Σ A: Type): Type := Σ -> Σ * A.

Definition bind (Σ: Type):
  forall (A B: Type) (f: M Σ A) (g: A -> M Σ B), M Σ B :=
  fun A B f g s1 =>
    match f s1 with
    | (s2, a) => g a s2
    end.

Definition ret (Σ: Type):
  forall (A: Type) (a: A), M Σ A :=
  fun A a s => (s, a).

End StateMonad.
```

```
#[export] Instance state_monad (Σ: Type): Monad (StateMonad.M Σ) := {|
  bind := StateMonad.bind Σ;
  ret := StateMonad.ret Σ;
|}.
```

以下是一些集合单子的例子

- 任取一个整数：

```
Definition any_Z: SetMonad.M Z := Sets.full.
```

- 整数乘二：

```
Definition multi_two: Z -> SetMonad.M Z :=
  fun x => ret (x * 2).
```

- 整数加一：

```
Definition plus_one: Z -> SetMonad.M Z :=
  fun x => ret (x + 1).
```

- 任取整数再乘二：

```
Definition bind_ex0: SetMonad.M Z :=
  bind any_Z multi_two.
```

- 任取整数乘二再加一：

```
Definition bind_ex1: SetMonad.M Z :=
  bind (bind any_Z multi_two) plus_one.
```

```
Definition bind_ex2: SetMonad.M Z :=
  bind any_Z (fun x => bind (multi_two x) plus_one).
```

下面是用单子描述计算时常用的记号：

```
Notation "x <- c1 ;; c2" := (bind c1 (fun x => c2))
  (at level 61, c1 at next level, right associativity) : monad_scope.
```

```
Notation " x : T <- c1 ;; c2" :=(bind c1 (fun x : T => c2))
  (at level 61, c1 at next level, right associativity) : monad_scope.
```

```
Notation "' pat <- c1 ;; c2" :=
  (bind c1 (fun x => match x with pat => c2 end))
  (at level 61, pat pattern, c1 at next level, right associativity) : monad_scope.
```

```
Notation "e1 ;; e2" := (bind e1 (fun _: unit => e2))
  (at level 61, right associativity) : monad_scope.
```

用这些 Notation 可以重写前面的一些例子。

- 任取整数再乘二：

```
Definition bind_ex0': SetMonad.M Z :=
  x <- any_Z;; ret (x * 2).
```

- 任取整数乘二再加一：

```
Definition bind_ex1': SetMonad.M Z :=
  x <- any_Z;; y <- multi_two x;; ret (y + 1).
```

3 集合单子的额外算子

```
Definition choice {A: Type} (f g: SetMonad.M A):
  SetMonad.M A :=
  f ∪ g.
```

```
Definition test (P: Prop): SetMonad.M unit :=
  fun _ => P.
```

```
Definition compute_abs (z: Z): SetMonad.M Z :=
  choice
    (test (z >= 0);; ret z)
    (test (z <= 0);; ret (-z)).
```

下面证明一些集合单子算子的性质
复合算子具有单调性：

```
#[export] Instance bind_mono (A B: Type):
  Proper (Sets.included ==> Sets.included ==> Sets.included)
    (@bind _ set_monad A B).
Proof.
  unfold Proper, respectful.
  unfold set_monad, bind, SetMonad.bind;
  sets_unfold; intros f1 f2 Hf g1 g2 Hg.
  intros b [a ?]; exists a.
  specialize (Hf a); specialize (Hg a b).
  tauto.
Qed.
```

复合算子保持集合相等：

```
#[export] Instance bind_congr (A B: Type):
  Proper (Sets.equiv ==> Sets.equiv ==> Sets.equiv)
    (@bind _ set_monad A B).
Proof.
  unfold Proper, respectful.
  unfold set_monad, bind, SetMonad.bind;
  sets_unfold; intros f1 f2 Hf g1 g2 Hg.
  intros b; split; intros [a ?]; exists a.
  + specialize (Hf a); specialize (Hg a b).
    tauto.
  + specialize (Hf a); specialize (Hg a b).
    tauto.
Qed.
```

复合算子具有对并集的分配律：

```
Lemma bind_union_distr_l:
  forall A B (f: SetMonad.M A) (g1 g2: A -> SetMonad.M B),
    bind f (g1 ∪ g2) == bind f g1 ∪ bind f g2.

Lemma bind_union_distr_r:
  forall A B (f1 f2: SetMonad.M A) (g: A -> SetMonad.M B),
    bind (f1 ∪ f2) g == bind f1 g ∪ bind f2 g.

Lemma bind_indexed_union_distr_l:
  forall A B I (f: SetMonad.M A) (g: I -> A -> SetMonad.M B),
    bind f (⋃ g) == ⋃ (fun i: I => bind f (g i)).

Lemma bind_indexed_union_distr_r:
  forall A B I (f: I -> SetMonad.M A) (g: A -> SetMonad.M B),
    bind (⋃ f) g == ⋃ (fun i: I => bind (f i) g).
```

习题 1. 复合算子具有结合律：

```
Lemma bind_assoc:
  forall (A B C: Type)
         (f: SetMonad.M A)
         (g: A -> SetMonad.M B)
         (h: B -> SetMonad.M C),
  bind (bind f g) h ==
  bind f (fun a => bind (g a) h).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2. 复合算子的左单位元是 ret：

```
Lemma bind_ret_l:
  forall (A B: Type)
         (a: A)
         (f: A -> SetMonad.M B),
  bind (ret a) f == f a.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 3. 复合算子的右单位元是 ret：

```
Lemma bind_ret_r:
  forall (A: Type)
         (f: SetMonad.M A),
  bind f ret == f.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

如果要表示带循环的计算过程，那就需要引入新的循环算子。
首先定义循环体的计算结果，其结果要么是 continue 终止，要么是 break 终止。

```
Inductive ContinueOrBreak (A B: Type): Type :=
| by_continue (a: A)
| by_break (b: B).
```

下面用不动点定义 repeat 循环。

```
Definition repeat_break_f
             {A B: Type}
             (body: A -> SetMonad.M (ContinueOrBreak A B))
             (W: A -> SetMonad.M B)
             (a: A): SetMonad.M B :=
  x <- body a;;
  match x with
  | by_continue a' => W a'
  | by_break b => ret b
  end.
```

```
Definition repeat_break
             {A B: Type}
             (body: A -> SetMonad.M (ContinueOrBreak A B)):
  A -> SetMonad.M B :=
  Kleene_LFix (repeat_break_f body).
```

下面证明 `repeat_break_f` 是单调连续的，从而表面上述不动点定理的应用是合理的。

```
Lemma repeat_break_unroll1:
  forall {A B: Type}
         (body: A -> SetMonad.M (ContinueOrBreak A B))
         (a: A),
    repeat_break body a ==
    x <- body a;;
    match x with
    | by_continue a' => repeat_break body a'
    | by_break b => ret b
    end.
```

下面还可以定义循环体中的 continue 语句和 break 语句。

```
Definition continue {A B: Type} (a: A):
  SetMonad.M (ContinueOrBreak A B) :=
  ret (by_continue a).
```

```
Definition break {A B: Type} (b: B):
  SetMonad.M (ContinueOrBreak A B) :=
  ret (by_break b).
```

4 单子上的霍尔逻辑

集合单子上，霍尔三元组会退化为霍尔二元组。

```
Definition Hoare {A: Type} (c: SetMonad.M A) (P: A -> Prop): Prop :=
  forall a, a ∈ c -> P a.
```

可以证明，各个单子算子满足下面性质。

```
Theorem Hoare_bind {A B: Type}:
  forall (f: SetMonad.M A)
         (g: A -> SetMonad.M B)
         (P: A -> Prop)
         (Q: B -> Prop),
    Hoare f P ->
    (forall a, P a -> Hoare (g a) Q) ->
    Hoare (bind f g) Q.
```

```
Theorem Hoare_ret {A: Type}:
  forall (a: A) (P: A -> Prop),
    P a -> Hoare (ret a) P.
```

```
Theorem Hoare_conseq {A: Type}:
  forall (f: SetMonad.M A) (P Q: A -> Prop),
    (forall a, P a -> Q a) ->
    Hoare f P ->
    Hoare f Q.
```

```
Theorem Hoare_conjunct {A: Type}:
  forall (f: SetMonad.M A) (P Q: A -> Prop),
    Hoare f P ->
    Hoare f Q ->
    Hoare f (fun a => P a /\ Q a).
```

```
Theorem Hoare_choice {A: Type}:
  forall (f g: SetMonad.M A)
         (P: A -> Prop),
    Hoare f P ->
    Hoare g P ->
    Hoare (choice f g) P.
```

```
Theorem Hoare_test_bind {A: Type}:
  forall (P: Prop)
         (f: SetMonad.M A)
         (Q: A -> Prop),
    (P -> Hoare f Q) ->
    (Hoare (test P;; f) Q).
```

```
Theorem Hoare_repeat_break {A B: Type}:
  forall (body: A -> SetMonad.M (ContinueOrBreak A B))
         (P: A -> Prop)
         (Q: B -> Prop),
    (forall a, P a ->
               Hoare (body a) (fun x => match x with
                                        | by_continue a => P a
                                        | by_break b => Q b
                                        end)) ->
    (forall a, P a -> Hoare (repeat_break body a) Q).
```

5 程序验证案例一：3x + 1

```
Definition body_3x1 (x: Z): SetMonad.M (ContinueOrBreak Z Z) :=
  choice
    (test (x <= 1);; break x)
    (choice
      (test (exists k, x = 2 * k);;
       continue (x / 2))
      (test (exists k, k <> 0 /\ x = 2 * k + 1);;
       continue (3 * x + 1))).
```

```
Definition run_3x1: Z -> SetMonad.M Z :=
  repeat_break body_3x1.
```

```
Theorem functional_correctness_3x1:
  forall n: Z,
    n >= 1 ->
    Hoare (run_3x1 n) (fun m => m = 1).
```

6 程序验证案例二：二分查找

```
Definition body_binary_search (P: Z -> Prop):
  Z * Z -> SetMonad.M (ContinueOrBreak (Z * Z) Z) :=
  fun '(lo, hi) =>
  choice
    (test (lo + 1 = hi);; break lo)
    (test (lo + 1 < hi);;
     let mid := (lo + hi) / 2 in
     choice
       (test (P mid);; continue (mid, hi))
       (test (~ P mid);; continue (lo, mid))).
```

```
Definition binary_search (P: Z -> Prop) (lo hi: Z):
  SetMonad.M Z :=
  repeat_break (body_binary_search P) (lo, hi).
```

```
Theorem functional_correctness_binary_search:
  forall (P: Z -> Prop) lo hi,
    (forall n m, n <= m -> P m -> P n) ->
    P lo ->
    ~ P hi ->
    Hoare (binary_search P lo hi)
          (fun x => P x /\ ~ P (x + 1)).
```